# Automatic Refunctionalization

## To a Language with Copattern Matching

With Applications to the Expression Problem

**Tillmann Rendel  Julia Trieflinger  Klaus Ostermann**

*University of Tübingen, Germany*

# Two Core Aspects of FP

# Two Core Aspects of FP

*programming with*
*first-class functions*


infinite behavior
black box
to use, apply to values

# Two Core Aspects of FP

*programming with*
*first-class functions*


infinite behavior
black box
to use, apply to values

# Two Core Aspects of FP

*programming with*
*first-class functions*

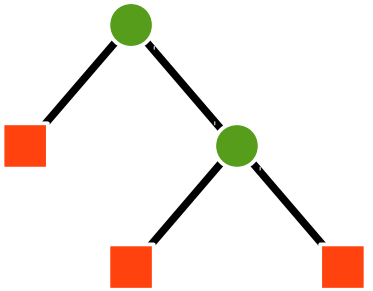

infinite behavior
black box
to use, apply to values

# Two Core Aspects of FP

*programming with first-class functions*

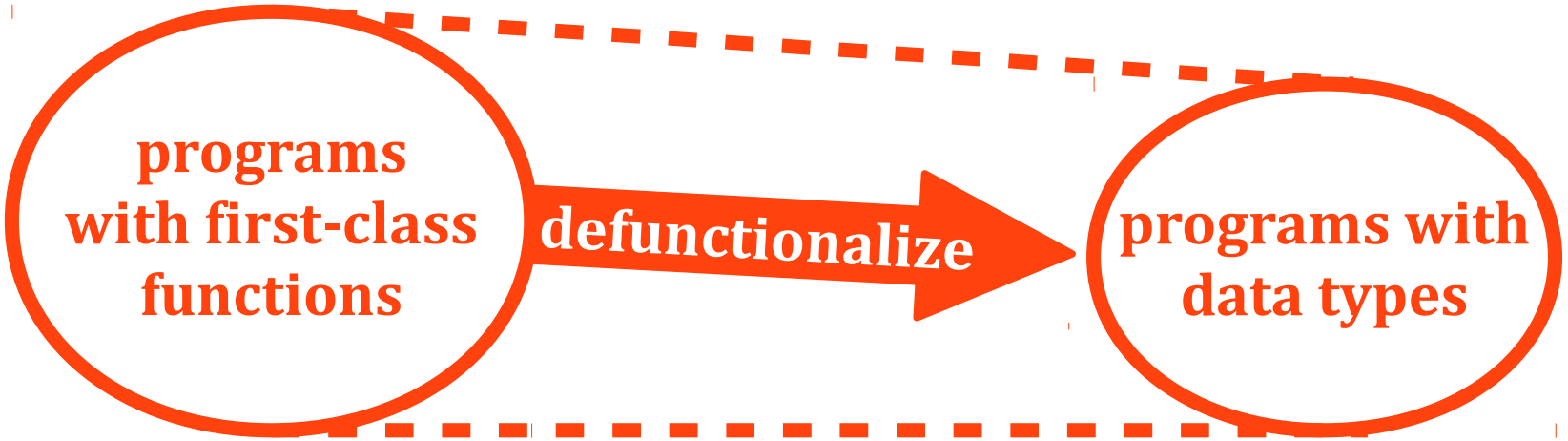

infinite behavior
black box
to use, apply to values

*programming with algebraic data types*



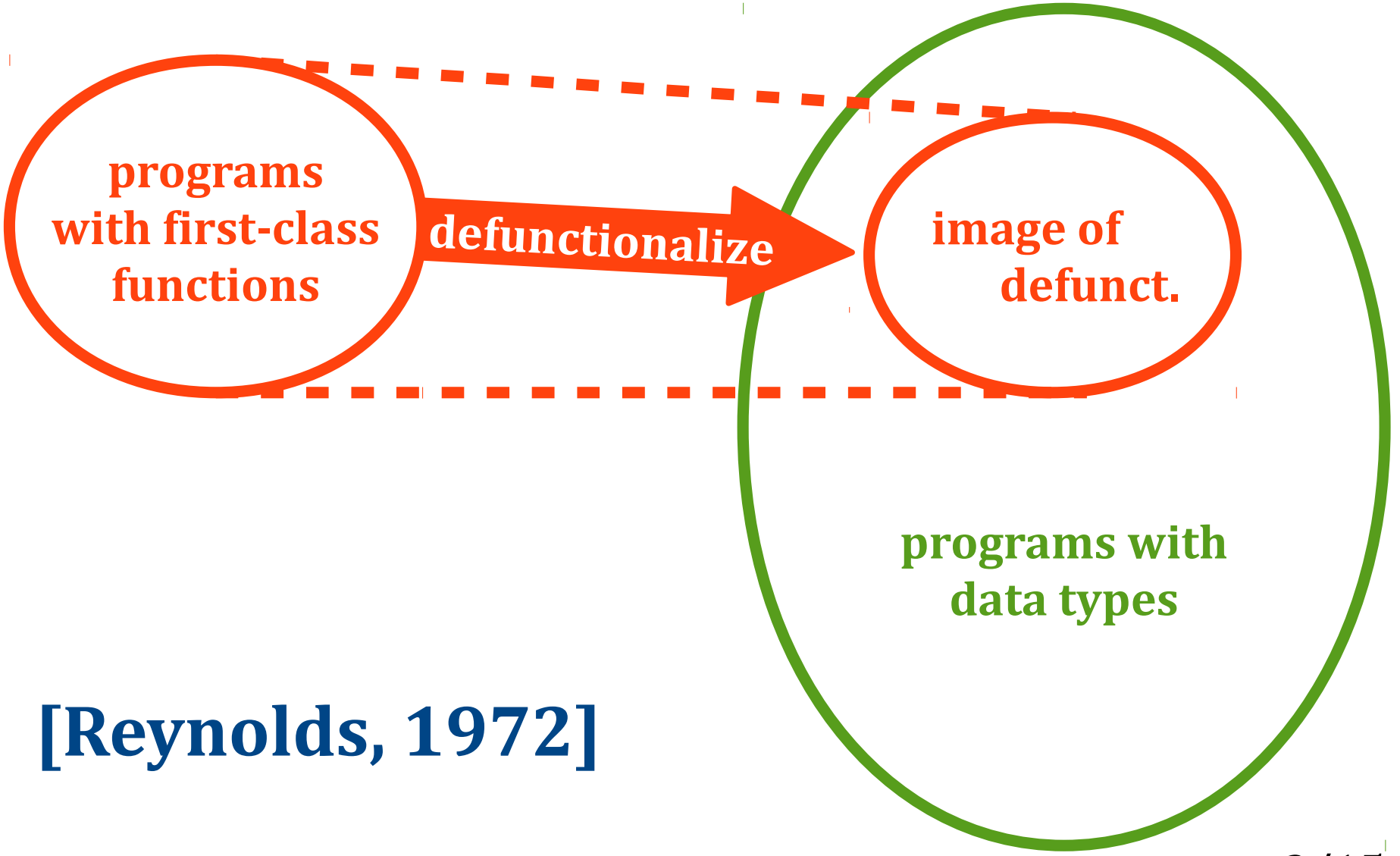
finite information
known structure
to use, traverse structure

# Defunctionalization



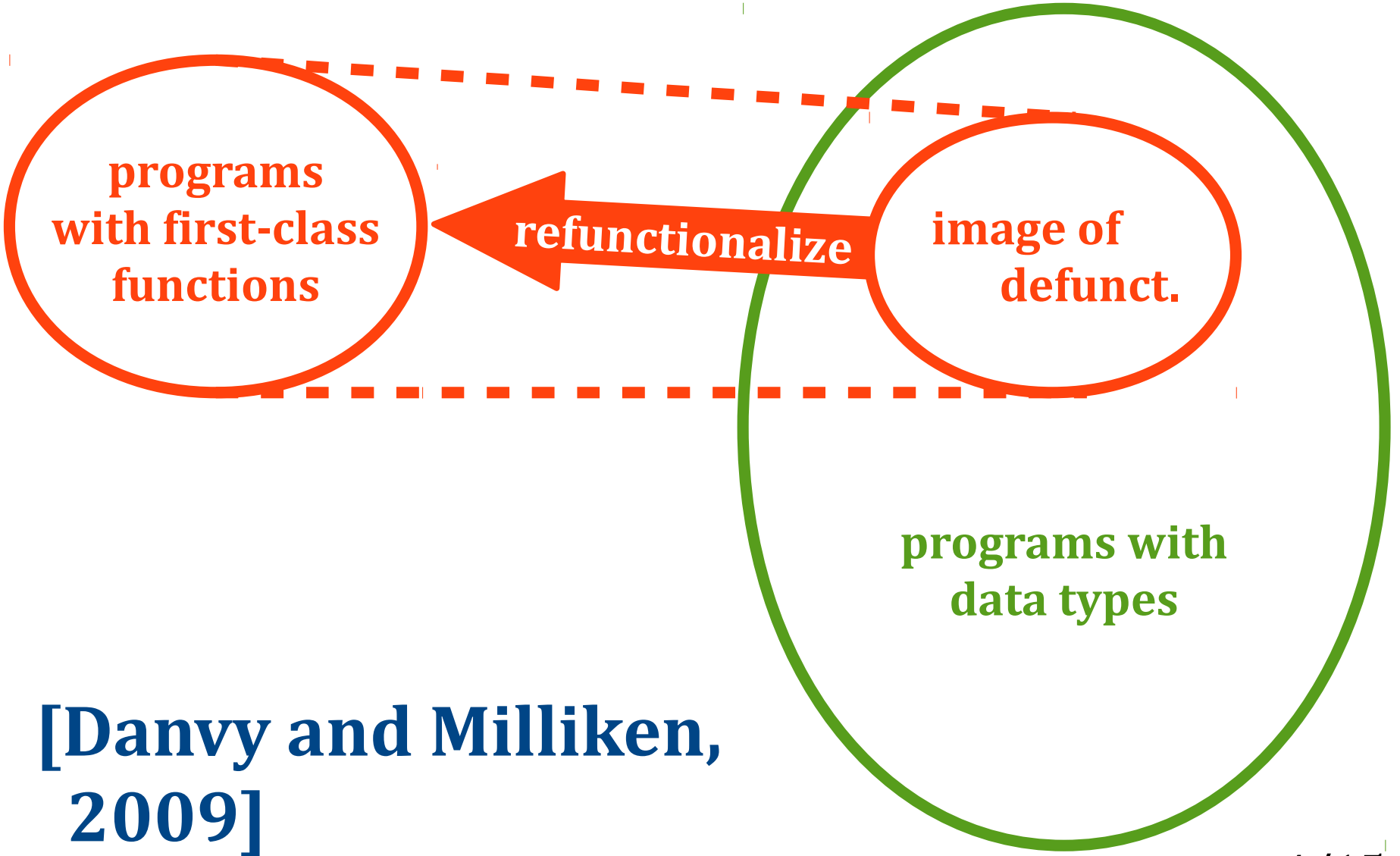

**[Reynolds, 1972]**

# Defunctionalization



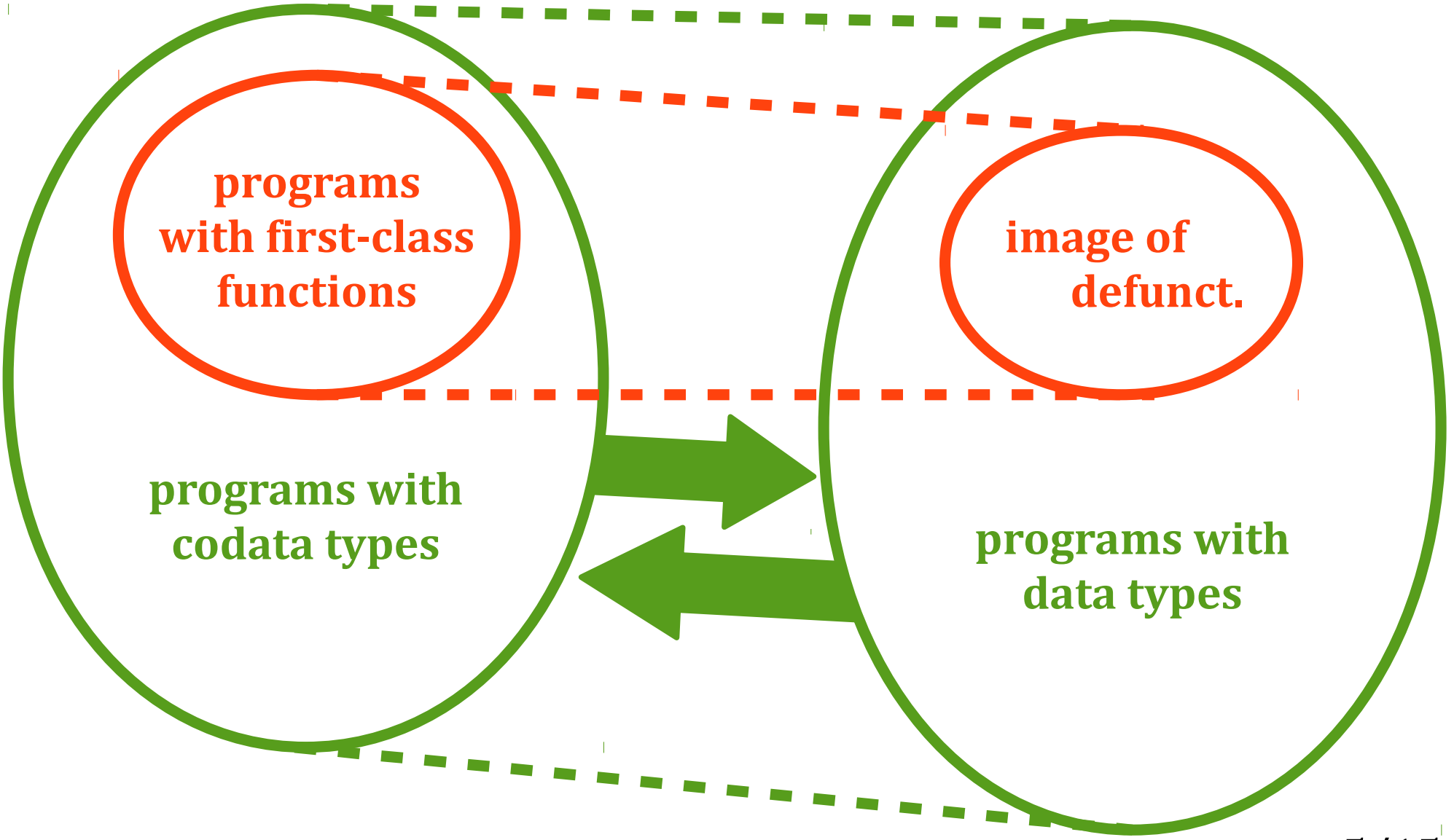

[Reynolds, 1972]

# Refunctionalization




[Danvy and Milliken, 2009]

# Wanted: Symmetric Languages



# Language with Data Types

- data type declarations

```
data Nat where
  zero() : Nat
  succ(Nat) : Nat
```

- first-order functions matching on first argument

```
function add(Nat, Nat) : Nat where
  add(zero(), m) = m
  add(succ(n), m) = succ(add(n, m))
```

# Language with Data Types

- data type declarations

```
data Nat where
  zero() : Nat
  succ(Nat) : Nat
```

- first-order functions matching on first argument

```
function add(Nat, Nat) : Nat where
  add(zero(), m) = m
  add(succ(n), m) = succ(add(n, m))
```

# Language with Data Types

- data type declarations

```
data Nat where
  zero() : Nat
  succ(Nat) : Nat
```

- first-order functions matching on first argument

```
function add(Nat, Nat) : Nat where
  add(zero(), m) = m
  add(succ(n), m) = succ(add(n, m))
```

# Language with Data Types

- data type declarations

```
data Nat where
  zero() : Nat
  succ(Nat) : Nat
```

- first-order functions matching on first argument

```
function add(Nat, Nat) : Nat where
  add(zero(), m) = m
  add(succ(n), m) = succ(add(n, m))
```

# Language with Data Types

- data type declarations

```
data Nat where
  zero() : Nat
  succ(Nat) : Nat
```

- first-order functions matching on first argument

```
function add(Nat, Nat) : Nat where
  add(zero(), m) = m
  add(succ(n), m) = succ(add(n, m))
```

# Language with Data Types

- data type declarations

```
data Nat where
  zero() : Nat
  succ(Nat) : Nat
```

- first-order functions matching on first argument

```
function add(Nat, Nat) : Nat where
  add(zero(), m) = m
  add(succ(n), m) = succ(add(n, m))
```

# Language with Data Types

- data type declarations

```
data Nat where
  zero() : Nat
  succ(Nat) : Nat
```

- first-order functions matching on first argument

```
function add(Nat, Nat) : Nat where
  add(zero(), m) = m
  add(succ(n), m) = succ(add(n, m))
```

# Language with Data Types

- data type declarations

```
data Nat where
  zero() : Nat
  succ(Nat) : Nat
```

- first-order functions matching on first argument

```
function add(Nat, Nat) : Nat where
  add(zero(), m) = m
  add(succ(n), m) = succ(add(n, m))
```

# Language with Codata Types

- codata type declarations

```
codata Stream where
  Stream.head() : Nat
  Stream.tail() : Stream
```

- first-order functions with copattern matching

```
function repeat(Nat) : Stream where
  repeat(n).head() = n
  repeat(n).tail() = repeat(n)
```

# Language with Codata Types

- codata type declarations

```
codata Stream where
  Stream.head() : Nat
  Stream.tail() : Stream
```

- first-order functions with copattern matching

```
function repeat(Nat) : Stream where
  repeat(n).head() = n
  repeat(n).tail() = repeat(n)
```

# Language with Codata Types

- codata type declarations

```
codata Stream where
  Stream.head() : Nat
  Stream.tail() : Stream
```

- first-order functions with copattern matching

```
function repeat(Nat) : Stream where
  repeat(n).head() = n
  repeat(n).tail() = repeat(n)
```

# Language with Codata Types

- codata type declarations

```
codata Stream where
  Stream.head() : Nat
  Stream.tail() : Stream
```

- first-order functions with copattern matching

```
function repeat(Nat) : Stream where
  repeat(n).head() = n
  repeat(n).tail() = repeat(n)
```

# Language with Codata Types

- codata type declarations

```
codata Stream where
  Stream.head() : Nat
  Stream.tail() : Stream
```

- first-order functions with copattern matching

```
function repeat(Nat) : Stream where
  repeat(n).head() = n
  repeat(n).tail() = repeat(n)
```

# Language with Codata Types

- codata type declarations

```
codata Stream where
  Stream.head() : Nat
  Stream.tail() : Stream
```

- first-order functions with copattern matching

```
function repeat(Nat) : Stream where
  repeat(n).head() = n
  repeat(n).tail() = repeat(n)
```

# Language with Codata Types

- codata type declarations

```
codata Stream where
  Stream.head() : Nat
  Stream.tail() : Stream
```

- first-order functions with copattern matching

```
function repeat(Nat) : Stream where
  repeat(n).head() = n
  repeat(n).tail() = repeat(n)
```

# Language with Codata Types

- codata type declarations

```
codata Stream where
  Stream.head() : Nat
  Stream.tail() : Stream
```

- first-order functions with copattern matching

```
function repeat(Nat) : Stream where
  repeat(n).head() = n
  repeat(n).tail() = repeat(n)
```

# Language with Codata Types

- codata type declarations

```
codata Stream where
  Stream.head() : Nat
  Stream.tail() : Stream
```

- first-order functions with copattern matching

```
function repeat(Nat) : Stream where
  repeat(n).head() = n
  repeat(n).tail() = repeat(n)
```

# Encoding First-Class Functions

- destructors with arguments

```
codata Fun where
  Fun.apply(Nat) : Nat
```

- example: mapping over a stream

```
function map(Stream, Fun) : Stream where
  map(s, f).head() = f.apply(s.head())
  map(s, f).tail() = map(s.tail(), f)
```

# Encoding First-Class Functions

- destructors with arguments

```
codata Fun where
  Fun.apply(Nat) : Nat
```

- example: mapping over a stream

```
function map(Stream, Fun) : Stream where
  map(s, f).head() = f.apply(s.head())
  map(s, f).tail() = map(s.tail(), f)
```

# Encoding First-Class Functions

- destructors with arguments

```
codata Fun where
  Fun.apply(Nat) : Nat
```

- example: mapping over a stream

```
function map(Stream, Fun) : Stream where
  map(s, f).head() = f.apply(s.head())
  map(s, f).tail() = map(s.tail(), f)
```

# Encoding First-Class Functions

- destructors with arguments

```
codata Fun where
  Fun.apply(Nat) : Nat
```

- example: mapping over a stream

```
function map(Stream, Fun) : Stream where
  map(s, f).head() = f.apply(s.head())
  map(s, f).tail() = map(s.tail(), f)
```

# Encoding First-Class Functions

- destructors with arguments

```
codata Fun where
  Fun.apply(Nat) : Nat
```

- example: mapping over a stream

```
function map(Stream, Fun) : Stream where
  map(s, f).head() = f.apply(s.head())
  map(s, f).tail() = map(s.tail(), f)
```

# Encoding First-Class Functions

- destructors with arguments

```
codata Fun where
  Fun.apply(Nat) : Nat
```

- example: mapping over a stream

```
function map(Stream, Fun) : Stream where
  map(s, f).head() = f.apply(s.head())
  map(s, f).tail() = map(s.tail(), f)
```

# Encoding First-Class Functions

- destructors with arguments

```
codata Fun where
  Fun.apply(Nat) : Nat
```

- example: mapping over a stream

```
function map(Stream, Fun) : Stream where
  map(s, f).head() = f.apply(s.head())
  map(s, f).tail() = map(s.tail(), f)
```

# Encoding First-Class Functions

- destructors with arguments

```
codata Fun where
  Fun.apply(Nat) : Nat
```

- example: mapping over a stream

```
function map(Stream, Fun) : Stream where
  map(s, f).head() = f.apply(s.head())
  map(s, f).tail() = map(s.tail(), f)
```

# Encoding First-Class Functions

- destructors with arguments

```
codata Fun where
  Fun.apply(Nat) : Nat
```

- example: mapping over a stream

```
function map(Stream, Fun) : Stream where
  map(s, f).head() = f.apply(s.head())
  map(s, f).tail() = map(s.tail(), f)
```

# Encoding First-Class Functions

- destructors with arguments

```
codata Fun where
  Fun.apply(Nat) : Nat
```

- example: mapping over a stream

```
function map(Stream, Fun) : Stream where
  map(s, f).head() = f.apply(s.head())
  map(s, f).tail() = map(s.tail(), f)
```

# Encoding First-Class Functions

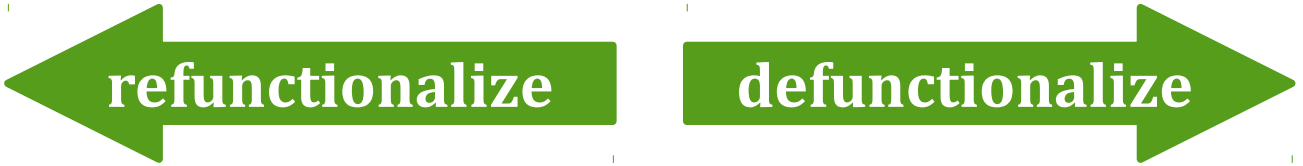
- destructors with arguments

```
codata Fun where
  Fun.apply(Nat) : Nat
```

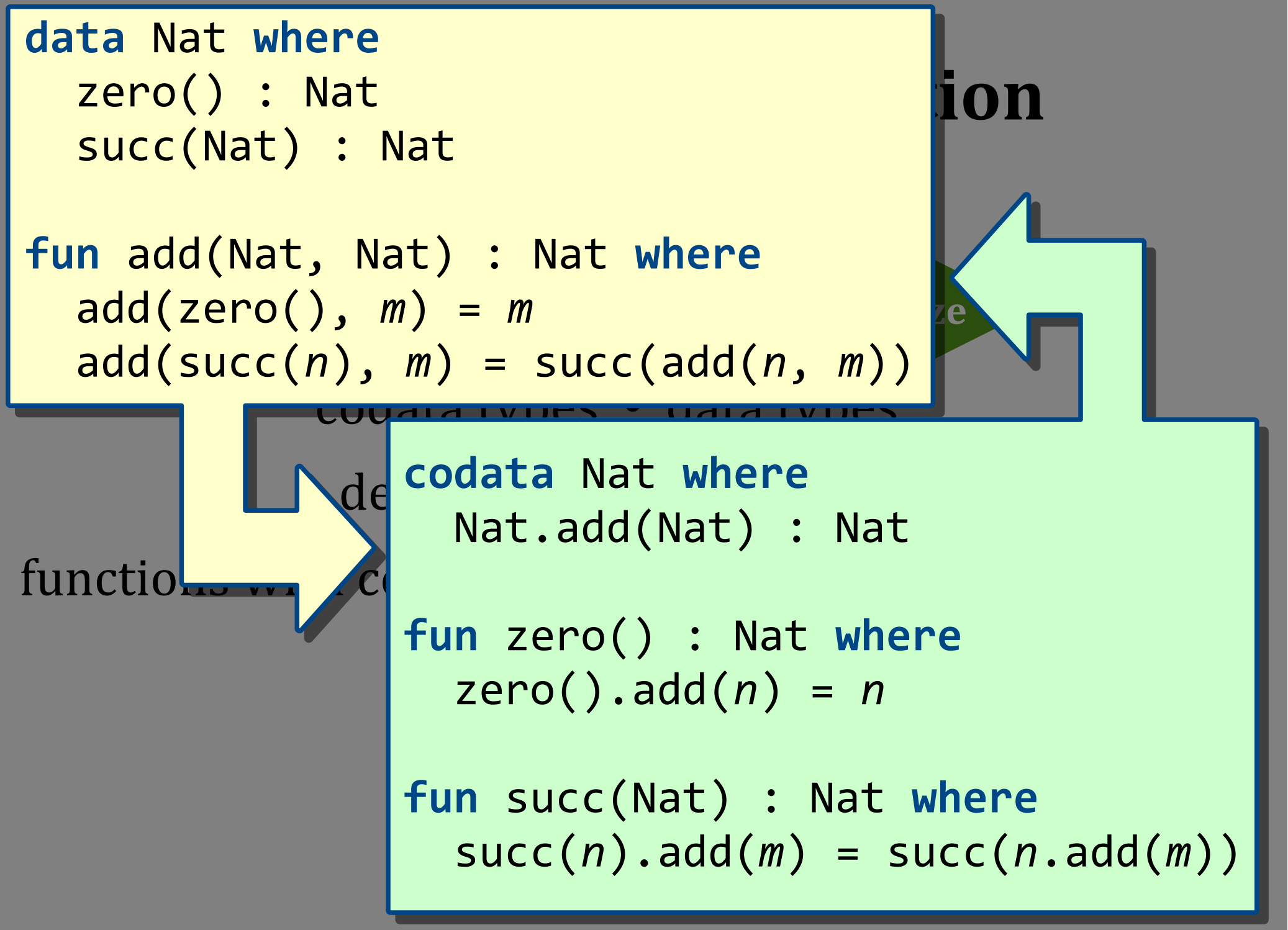
- example: mapping over a stream

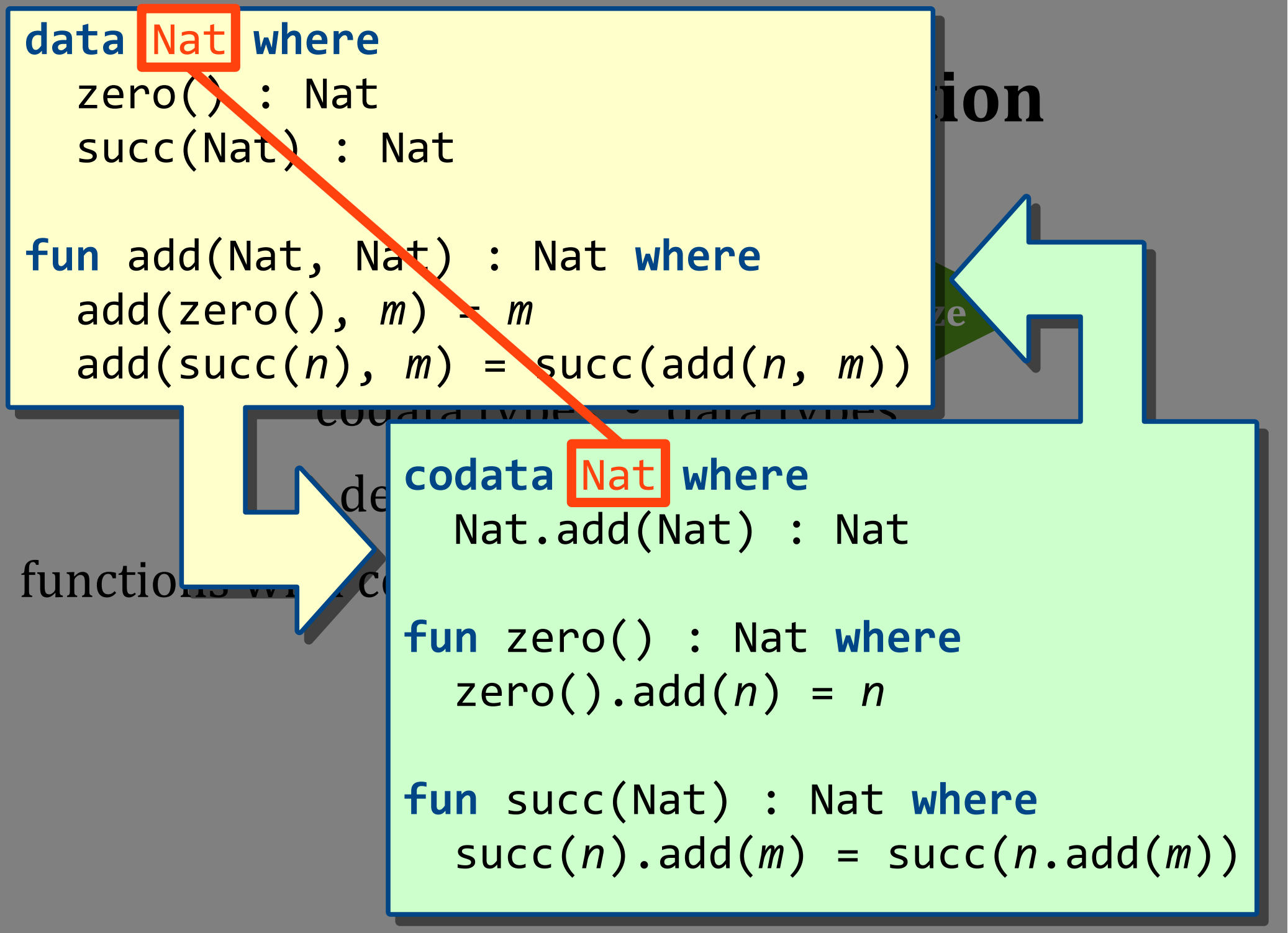
```
function map(Stream, Fun) : Stream where
  map(s, f).head() = f.apply(s.head())
  map(s, f).tail() = map(s.tail(), f)
```

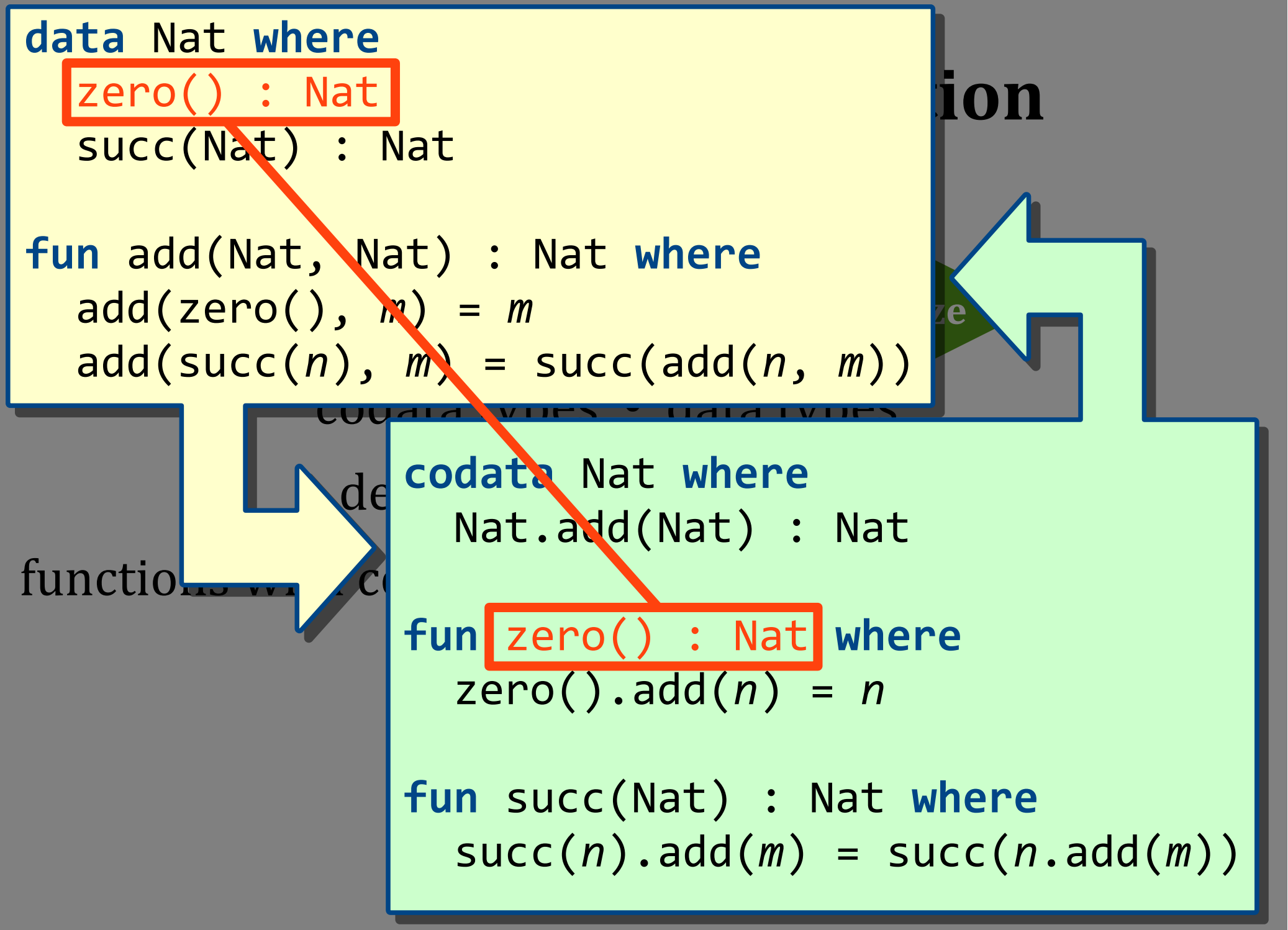
# (De|Re)functionalization

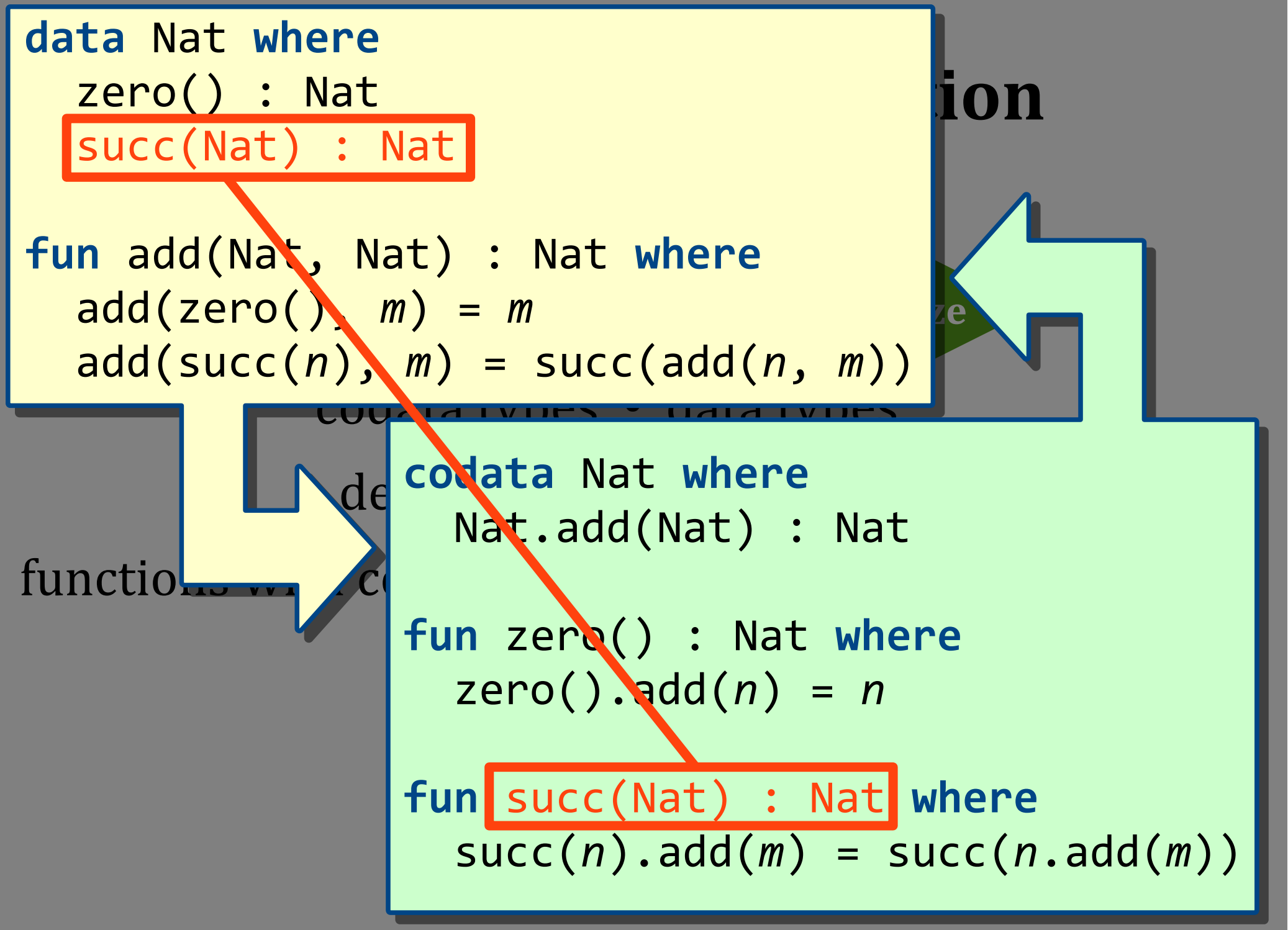


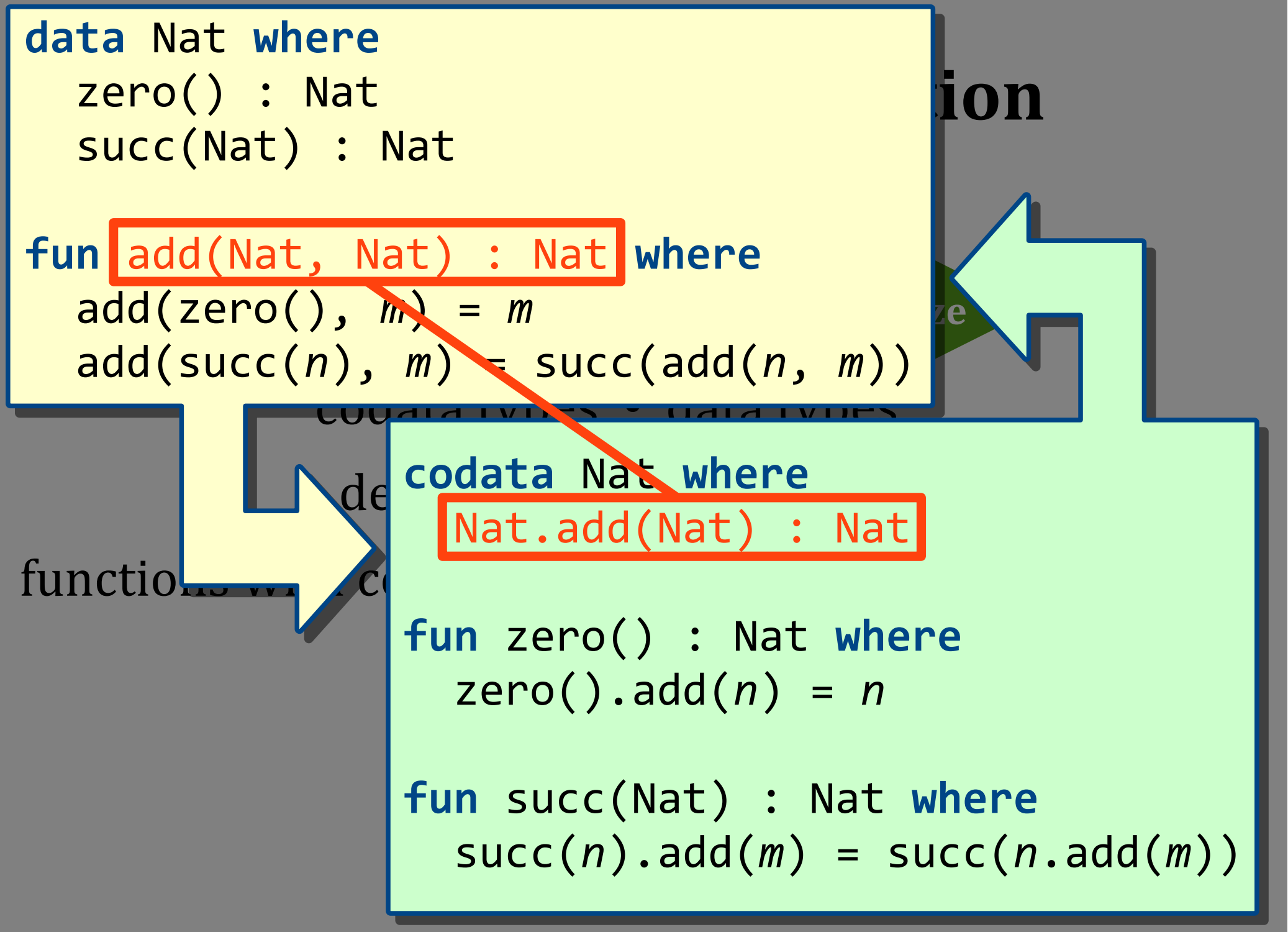

| | | |
|---|---|---|
| codata types | • | data types |
| destructors | • | functions with patterns |
| functions with copatterns | • | constructors |

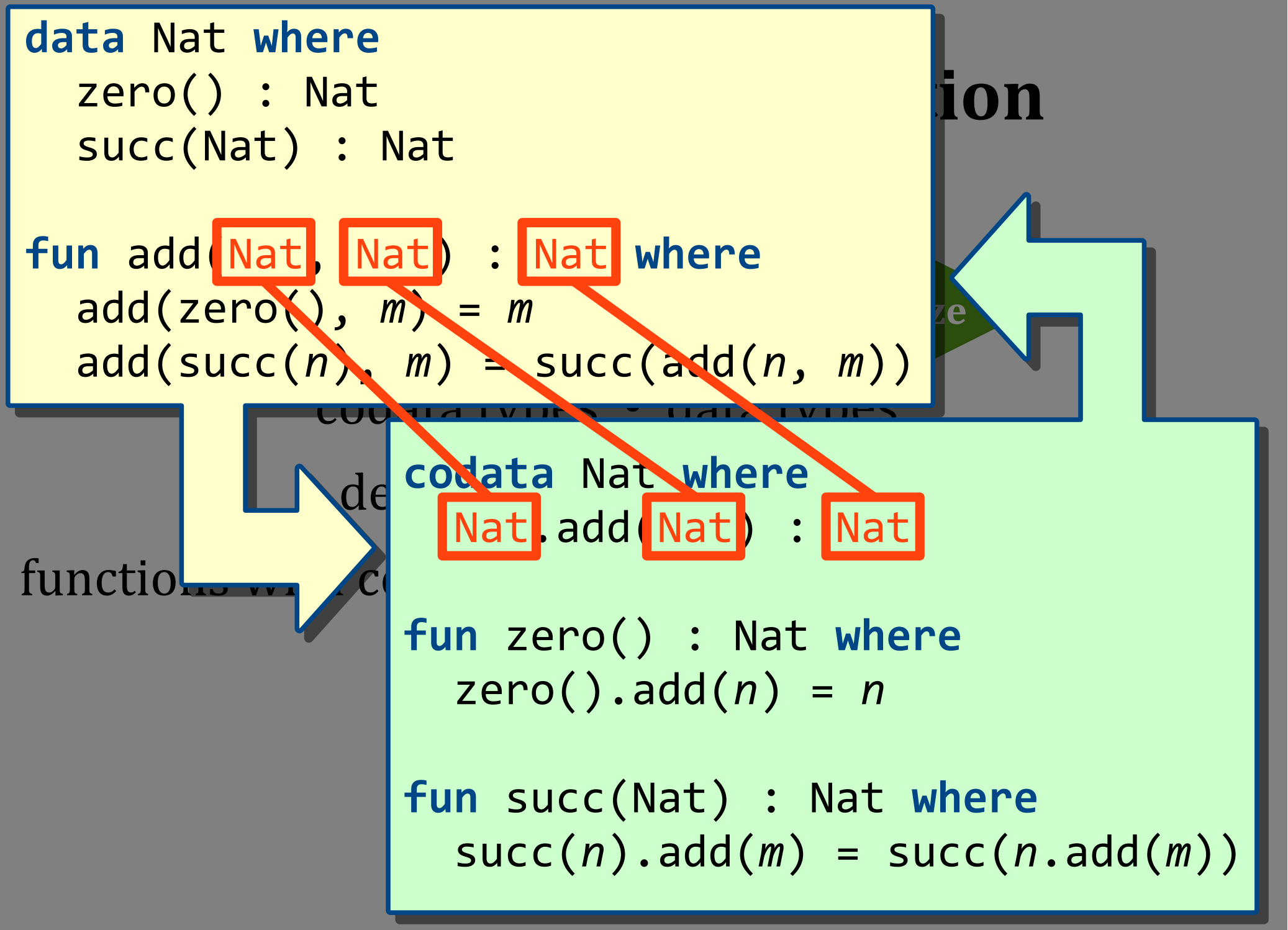

data Nat where
zero() : Nat
succ(Nat) : Nat
fun add(Nat, Nat) : Nat where
add(zero(), m) = m
add(succ(n), m) = succ(add(n, m))
codata Nat where
Nat.add(Nat) : Nat
fun zero() : Nat where
zero().add(n) = n
fun succ(Nat) : Nat where
succ(n).add(m) = succ(n.add(m))

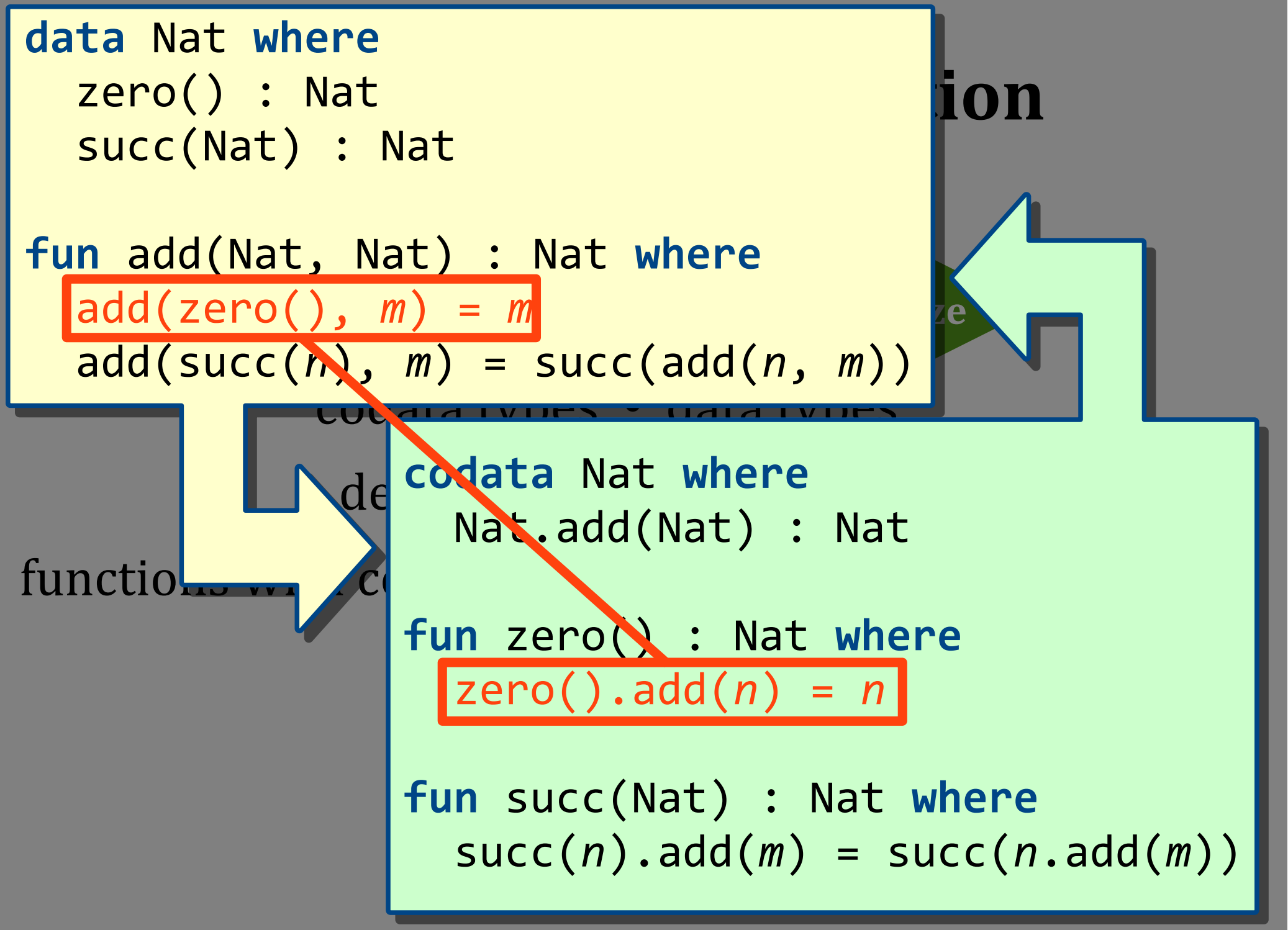

data Nat where
zero() : Nat
succ(Nat) : Nat
fun add(Nat, Nat) : Nat where
add(zero(), m) = m
add(succ(n), m) = succ(add(n, m))
codata Nat where
Nat.add(Nat) : Nat
fun zero() : Nat where
zero().add(n) = n
fun succ(Nat) : Nat where
succ(n).add(m) = succ(n.add(m))

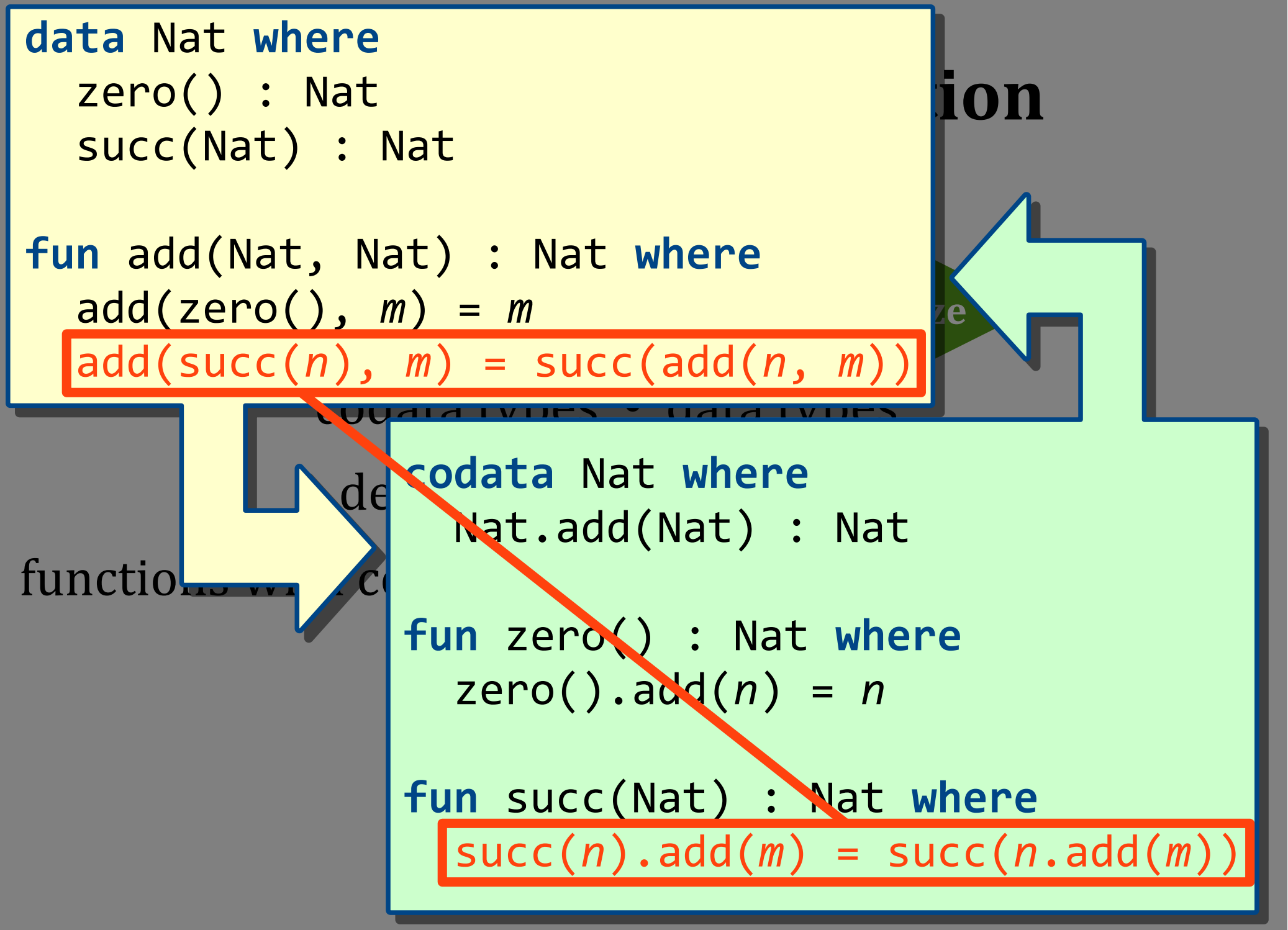

data Nat where
zero() : Nat
succ(Nat) : Nat
fun add(Nat, Nat) : Nat where
add(zero(), m) = m
add(succ(n), m) = succ(add(n, m))
codata Nat where
Nat.add(Nat) : Nat
fun zero() : Nat where
zero().add(n) = n
fun succ(Nat) : Nat where
succ(n).add(m) = succ(n.add(m))


data Nat where
zero() : Nat
succ(Nat) : Nat
fun add(Nat, Nat) : Nat where
add(zero(), m) = m
add(succ(n), m) = succ(add(n, m))
codata Nat where
Nat.add(Nat) : Nat
fun zero() : Nat where
zero().add(n) = n
fun succ(Nat) : Nat where
succ(n).add(m) = succ(n.add(m))

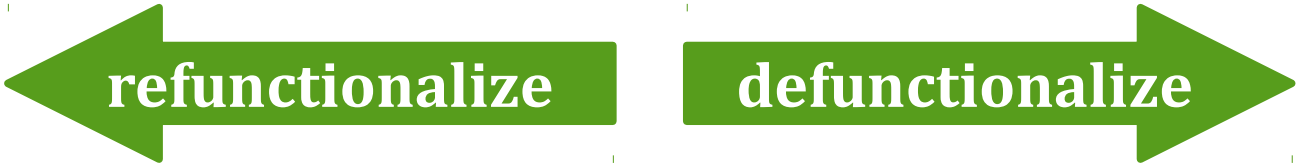

data Nat where
zero() : Nat
succ(Nat) : Nat
fun add(Nat, Nat) : Nat where
add(zero(), m) = m
add(succ(n), m) = succ(add(n, m))
codata Nat where
Nat.add(Nat) : Nat
fun zero() : Nat where
zero().add(n) = n
fun succ(Nat) : Nat where
succ(n).add(m) = succ(n.add(m))


data Nat where
zero() : Nat
succ(Nat) : Nat
fun add(Nat, Nat) : Nat where
add(zero(), m) = m
add(succ(n), m) = succ(add(n, m))
codata Nat where
Nat.add(Nat) : Nat
fun zero() : Nat where
zero().add(n) = n
fun succ(Nat) : Nat where
succ(n).add(m) = succ(n.add(m))


data Nat where
zero() : Nat
succ(Nat) : Nat
fun add(Nat, Nat) : Nat where
add(zero(), m) = m
add(succ(n), m) = succ(add(n, m))
codata Nat where
Nat.add(Nat) : Nat
fun zero() : Nat where
zero().add(n) = n
fun succ(Nat) : Nat where
succ(n).add(m) = succ(n.add(m))


data Nat where
zero() : Nat
succ(Nat) : Nat
fun add(Nat, Nat) : Nat where
add(zero(), m) = m
add(succ(n), m) = succ(add(n, m))
codata Nat where
Nat.add(Nat) : Nat
fun zero() : Nat where
zero().add(n) = n
fun succ(Nat) : Nat where
succ(n).add(m) = succ(n.add(m))

# (De|Re)functionalization




| | | |
|---|---|---|
| codata types | • | data types |
| destructors | • | functions with patterns |
| functions with copatterns | • | constructors |

# Automatic Refunctionalization

- Codata with multiple observations that take arguments
  - to support the full data language
- Top-level, first-order functions, copatterns
  - to avoid lambda lifting and name mangling

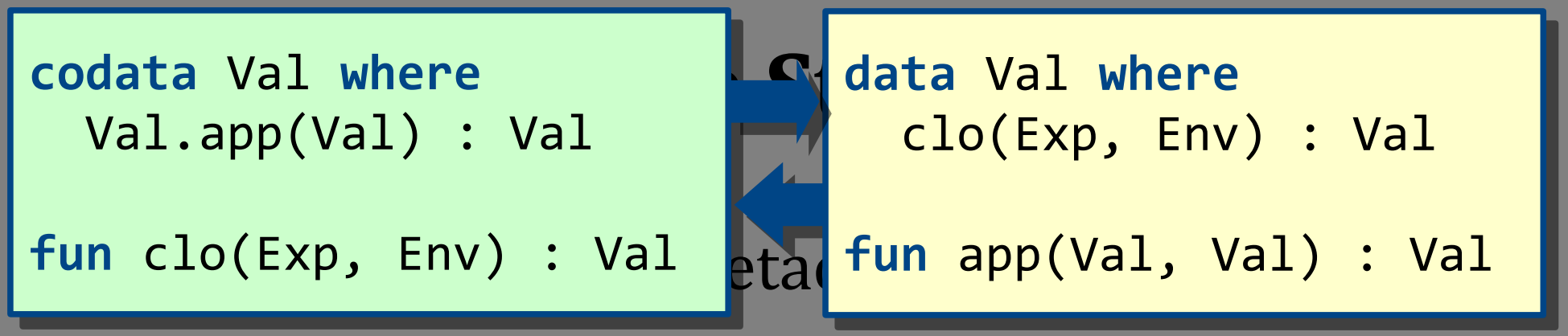
# Case Study

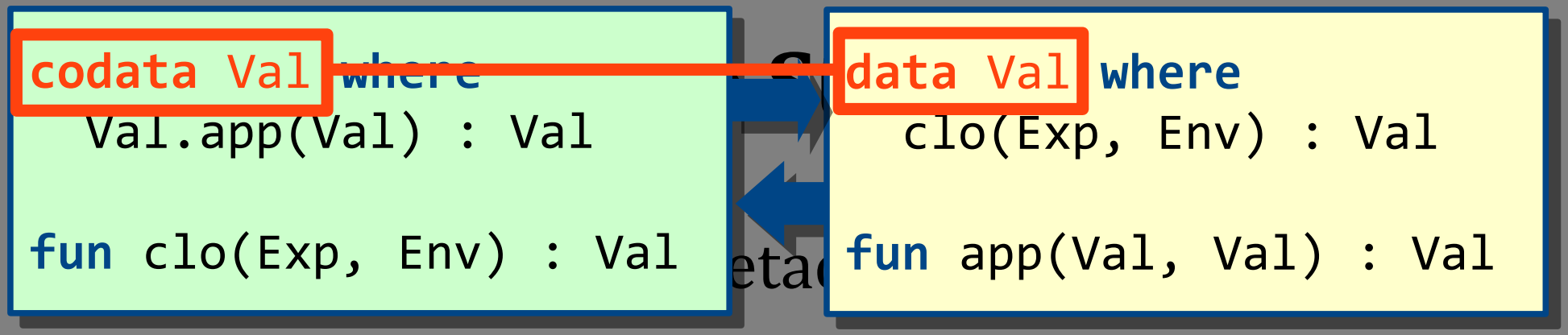
- Based on Reynold's metacircular interpreter

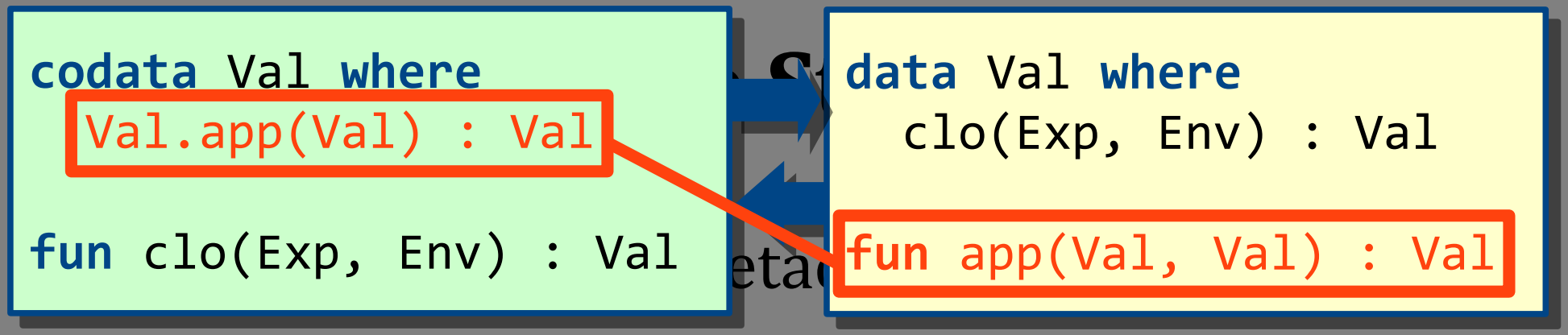
# e Study

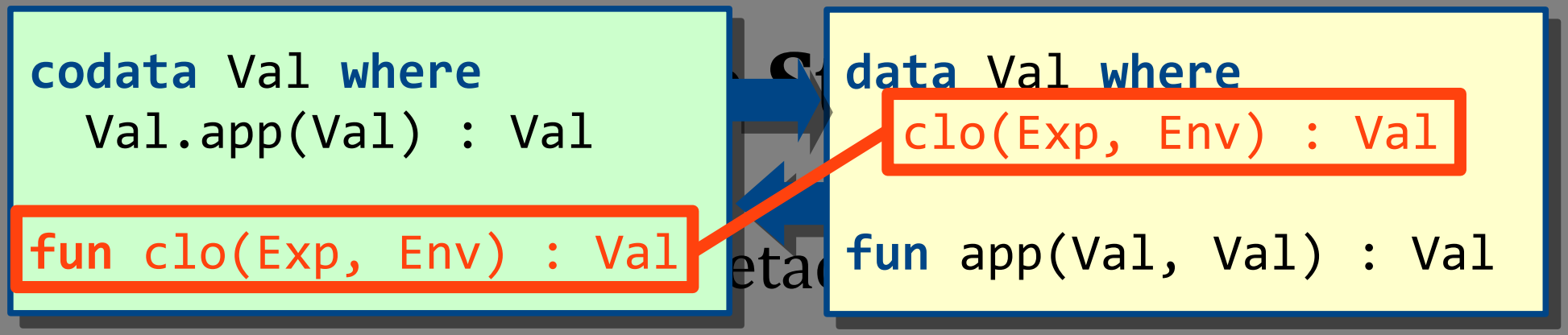
```
codata Val where
  Val.app(Val) : Val

fun clo(Exp, Env) : Val
```

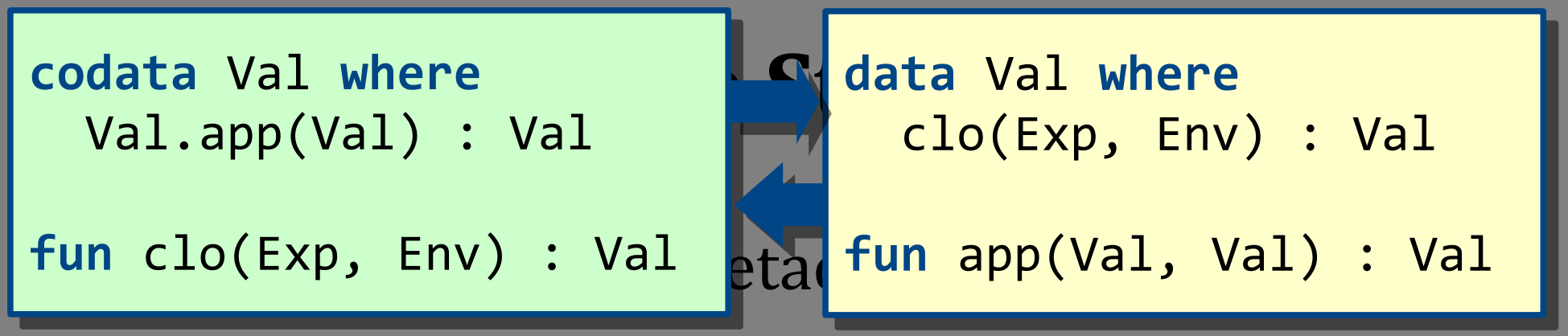
etacircular interpreter

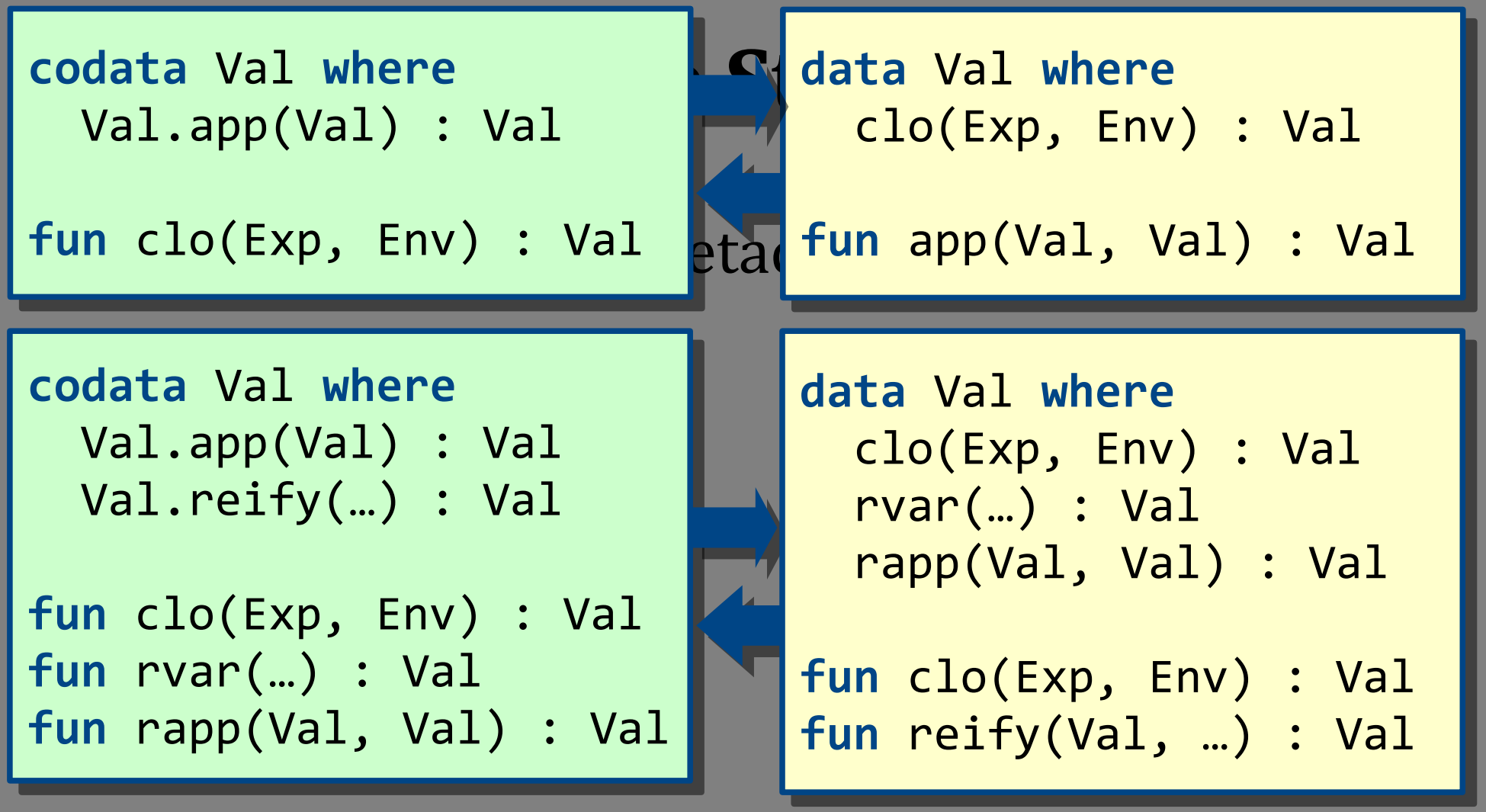

codata Val where
Val.app(Val) : Val
fun clo(Exp, Env) : Val
data Val where
clo(Exp, Env) : Val
fun app(Val, Val) : Val

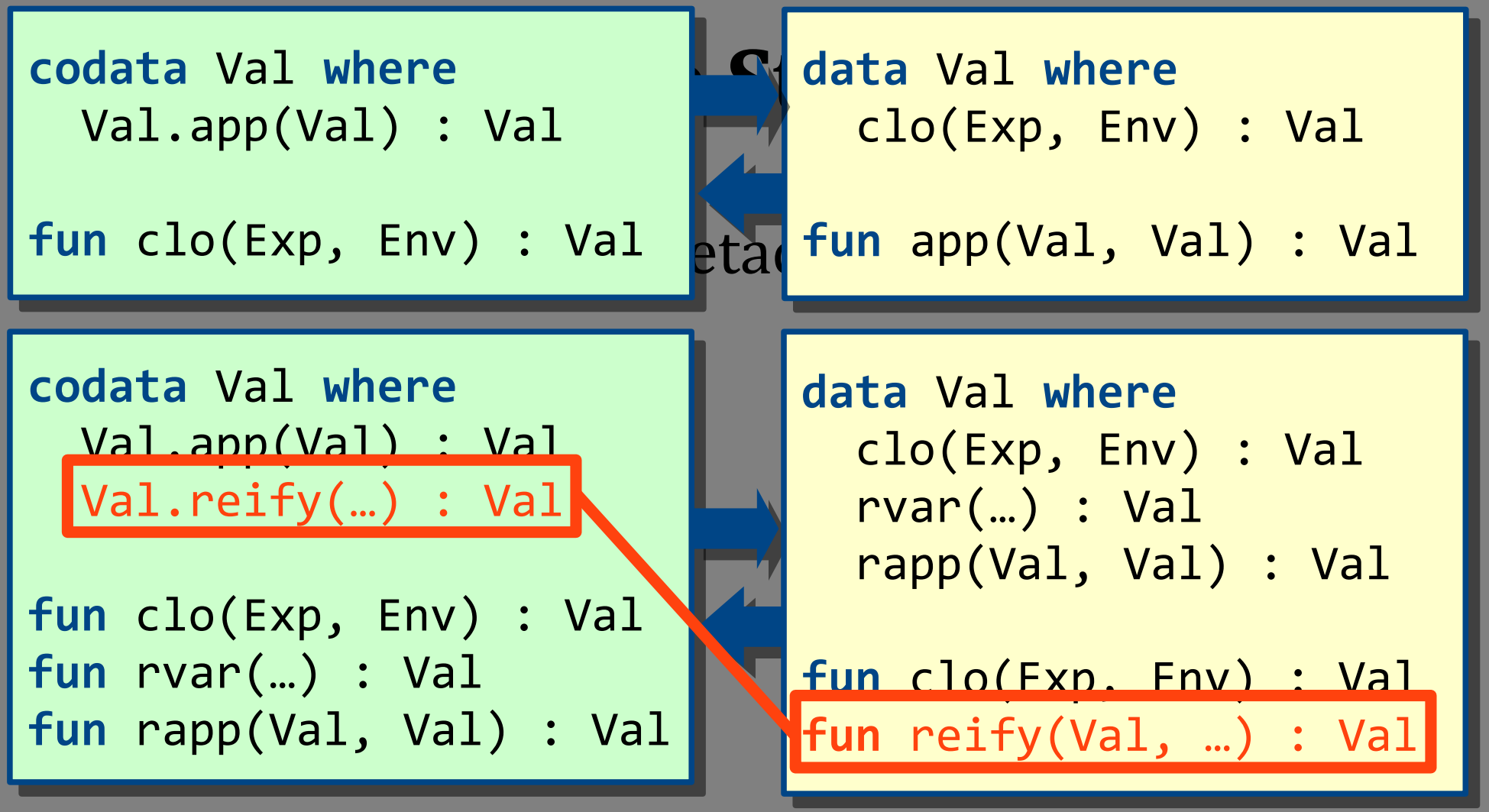

codata Val where
Val.app(Val) : Val
fun clo(Exp, Env) : Val
data Val where
clo(Exp, Env) : Val
fun app(Val, Val) : Val

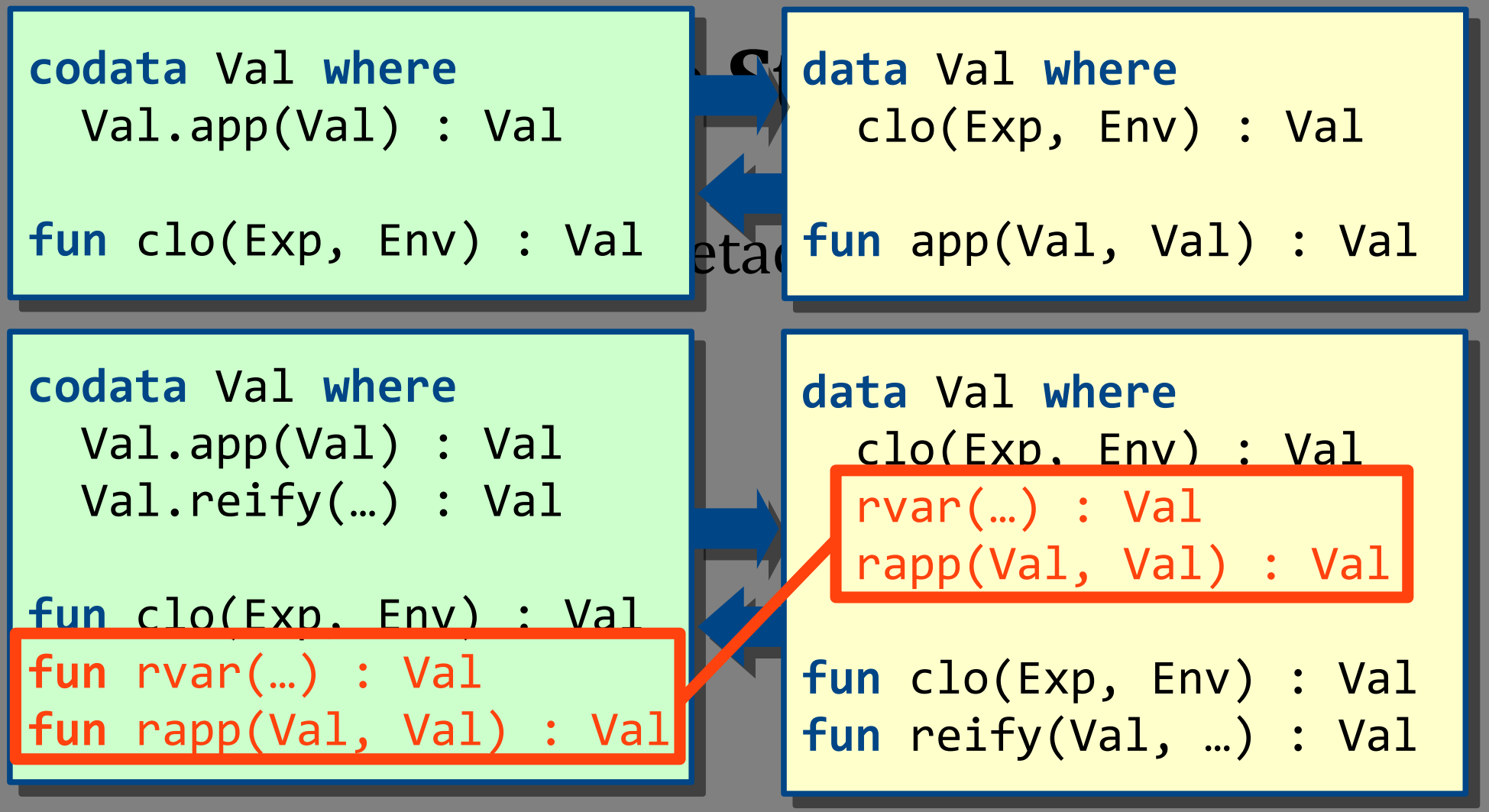

codata Val where
Val.app(Val) : Val
fun clo(Exp, Env) : Val
data Val where
clo(Exp, Env) : Val
fun app(Val, Val) : Val


codata Val where
Val.app(Val) : Val
fun clo(Exp, Env) : Val
data Val where
clo(Exp, Env) : Val
fun app(Val, Val) : Val


codata Val where
Val.app(Val) : Val
fun clo(Exp, Env) : Val
data Val where
clo(Exp, Env) : Val
fun app(Val, Val) : Val


codata Val where
Val.app(Val) : Val
fun clo(Exp, Env) : Val
data Val where
clo(Exp, Env) : Val
fun app(Val, Val) : Val
codata Val where
Val.app(Val) : Val
Val.reify(…) : Val
fun clo(Exp, Env) : Val
fun rvar(…) : Val
fun rapp(Val, Val) : Val
data Val where
clo(Exp, Env) : Val
rvar(…) : Val
rapp(Val, Val) : Val
fun clo(Exp, Env) : Val
fun reify(Val, …) : Val


codata Val where
Val.app(Val) : Val
fun clo(Exp, Env) : Val
data Val where
clo(Exp, Env) : Val
fun app(Val, Val) : Val
codata Val where
Val.reify(…) : Val
fun clo(Exp, Env) : Val
fun rvar(…) : Val
fun rapp(Val, Val) : Val
data Val where
clo(Exp, Env) : Val
rvar(…) : Val
rapp(Val, Val) : Val
fun reify(Val, …) : Val


codata Val where
Val.app(Val) : Val
fun clo(Exp, Env) : Val
data Val where
clo(Exp, Env) : Val
fun app(Val, Val) : Val
codata Val where
Val.app(Val) : Val
Val.reify(…) : Val
fun clo(Exp, Env) : Val
fun rvar(…) : Val
fun rapp(Val, Val) : Val
data Val where
clo(Exp, Env) : Val
rvar(…) : Val
rapp(Val, Val) : Val
fun clo(Exp, Env) : Val
fun reify(Val, …) : Val

```
codata Val where
  Val.app(Val) : Val

fun clo(Exp, Env) : Val
```

```
data Val where
  clo(Exp, Env) : Val

fun app(Val, Val) : Val
```

```
codata Val where
  Val.app(Val) : Val
  Val.reify(…) : Val

fun clo(Exp, Env) : Val
fun rvar(…) : Val
fun rapp(Val, Val) : Val
```

```
data Val where
  clo(Exp, Env) : Val
  rvar(…) : Val
  rapp(Val, Val) : Val

fun clo(Exp, Env) : Val
fun reify(Val, …) : Val
```

# Case Study

- Based on Reynold's metacircular interpreter
- Extension to Normalization by Evaluation
- Different extensions are modular
  in data vs. codata fragment
- Expression Problem?

# Programs as Matrices

```
data Nat where                  zero() : Nat            succ(Nat) : Nat

fun add(Nat, Nat) : Nat where   add(zero(), n) = n      add(succ(m), n) = succ(add(m, n))
```

```
codata Nat where                Nat.add(Nat) : Nat

fun zero() : Nat where          zero().add(n) = n

fun succ(Nat) : Nat where       succ(m).add(n) = succ(m.add(n))
```

# Programs as Matrices

| `data Nat where` | `zero() : Nat` | `succ(Nat) : Nat` |
|---|---|---|
| `fun add(Nat, Nat) : Nat where` | `add(zero(), n) = n` | `add(succ(m), n) = succ(add(m, n))` |

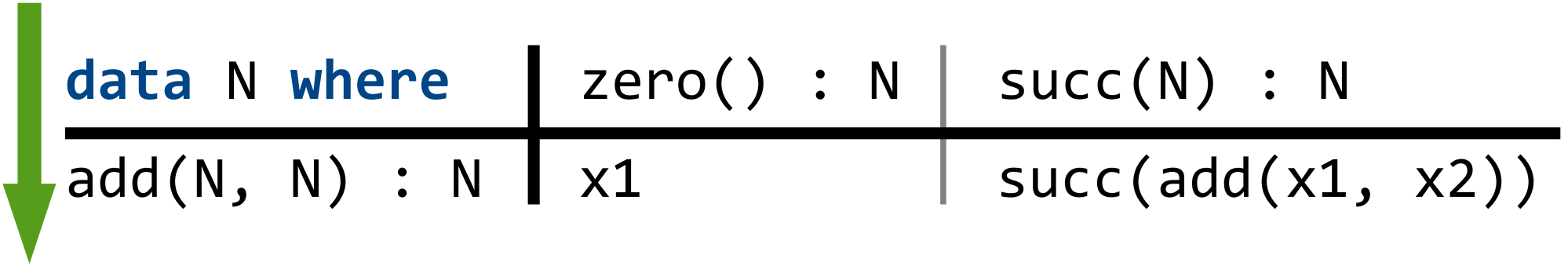
| `codata Nat where` | `Nat.add(Nat) : Nat` |
|---|---|
| `fun zero() : Nat where` | `zero().add(n) = n` |
| `fun succ(Nat) : Nat where` | `succ(m).add(n) = succ(m.add(n))` |

# Programs as Matrices

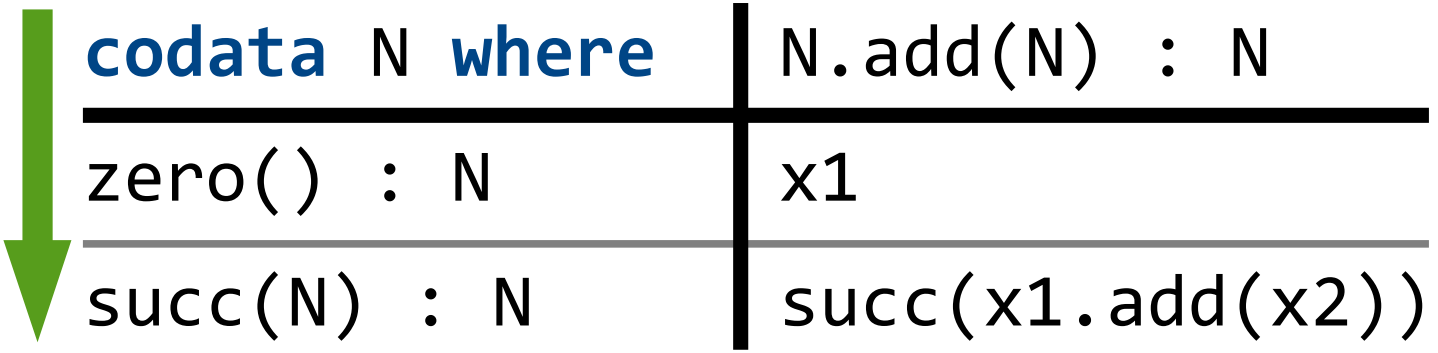
| `data N where` | `zero() : N` | `succ(N) : N` |
|---|---|---|
| `add(N, N) : N` | `x1` | `succ(add(x1, x2))` |

| `codata N where` | `N.add(N) : N` |
|---|---|
| `zero() : N` | `x1` |
| `succ(N) : N` | `succ(x1.add(x2))` |

# Programs as Matrices

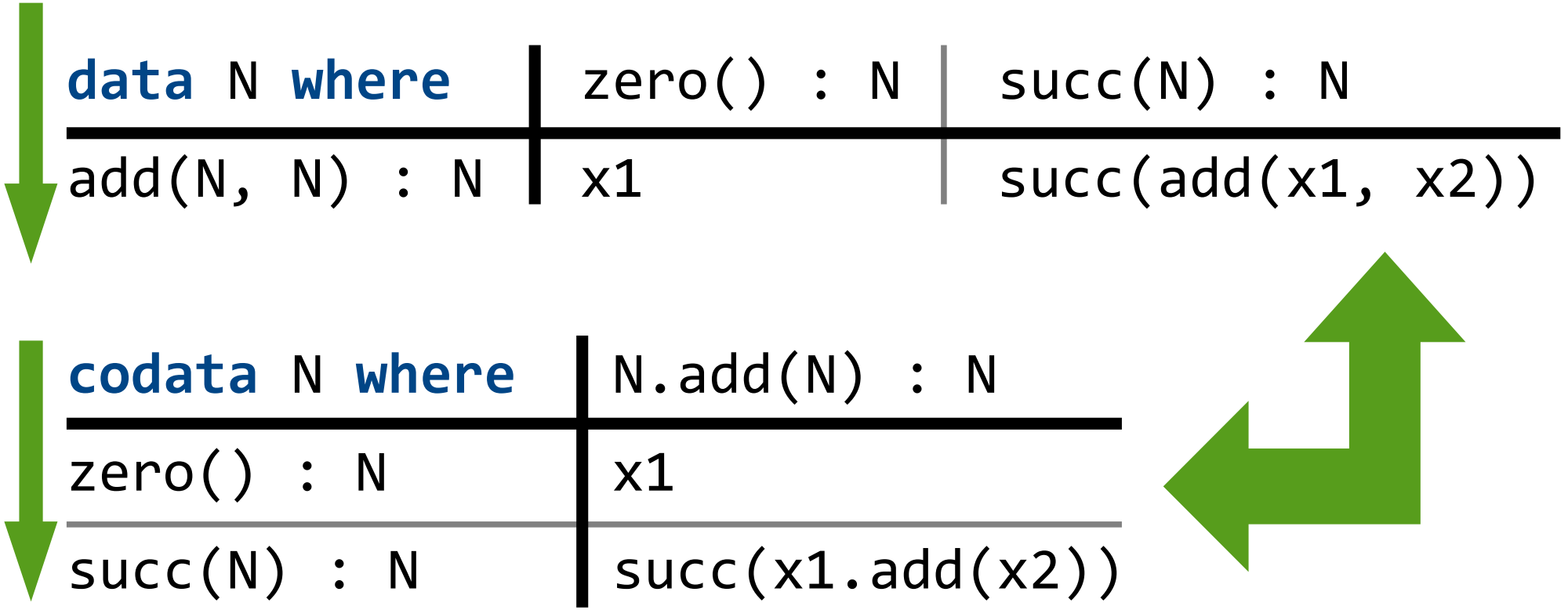


| `data N where` | `zero() : N` | `succ(N) : N` |
|---|---|---|
| `add(N, N) : N` | `x1` | `succ(add(x1, x2))` |



| `codata N where` | `N.add(N) : N` |
|---|---|
| `zero() : N` | `x1` |
| `succ(N) : N` | `succ(x1.add(x2))` |

# Programs as Matrices

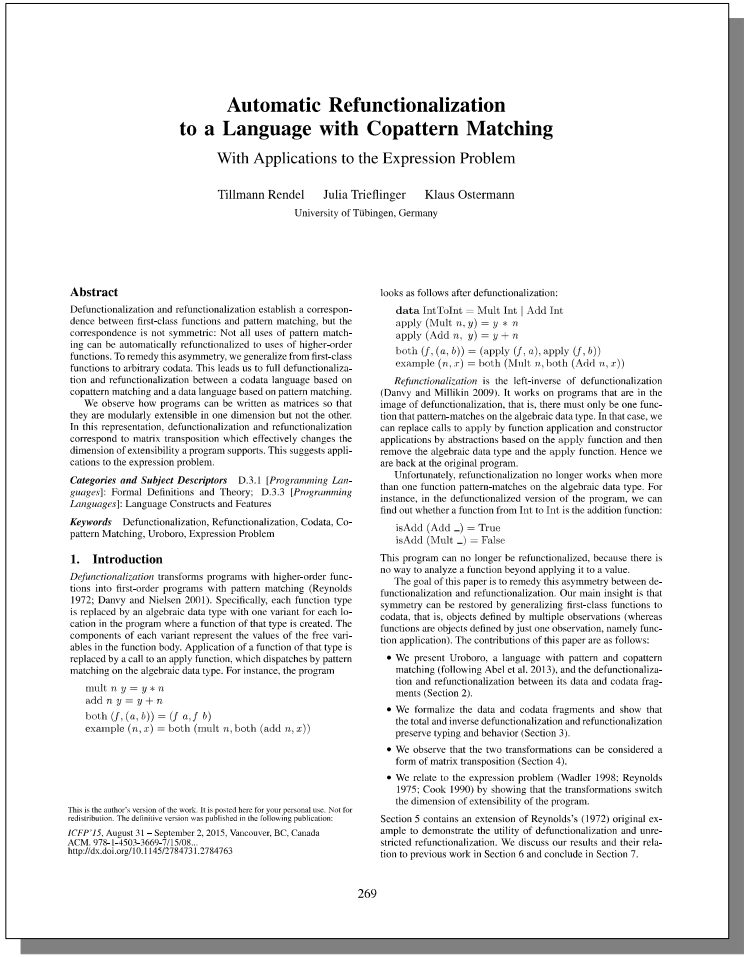


# Programs as Matrices

- Programs can be written as matrices
- (De|Re)functionalization is
  matrix transposition
- Modular extension by rows, not by columns

- (De|Re)functionalization changes the dimension of modularly supported extensibility

# In The Paper



**Automatic Refunctionalization to a Language with Copattern Matching**

With Applications to the Expression Problem

Tillmann Rendel Julia Trieflinger Klaus Ostermann
University of Tübingen, Germany

**Abstract**

Defunctionalization and refunctionalization establish a correspondence between first-class functions and pattern matching, but the correspondence is not symmetric: Not all uses of pattern matching can be automatically refunctionalized to uses of higher-order functions. To remedy this asymmetry, we generalize from first-class functions to arbitrary codata. This leads us to full defunctionalization and refunctionalization between a codata language based on copattern matching and a data language based on pattern matching.

We observe how programs can be written as matrices so that they are modularly extensible in one dimension but not the other. In this representation, defunctionalization and refunctionalization correspond to matrix transposition which effectively changes the dimension of extensibility a program supports. This suggests applications to the expression problem.

***Categories and Subject Descriptors*** D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features

***Keywords*** Defunctionalization, Refunctionalization, Codata, Copattern Matching, Uroboro, Expression Problem

**1. Introduction**

*Defunctionalization* transforms programs with higher-order functions into first-order programs with pattern matching (Reynolds 1972; Danvy and Nielsen 2001). Specifically, each function type is replaced by an algebraic data type with one variant for each location in the program where a function of that type is created. The components of each variant represent the values of the free variables in the function body. Application of a function of that type is replaced by a call to an apply function, which dispatches by pattern matching on the algebraic data type. For instance, the program

$$\begin{aligned}&\text{mult } n\ y = y * n\\&\text{add } n\ y = y + n\\&\text{both } (f, (a, b)) = (f\ a, f\ b)\\&\text{example } (n, x) = \text{both } (\text{mult } n, \text{both } (\text{add } n, x))\end{aligned}$$

looks as follows after defunctionalization:

$$\begin{aligned}&\textbf{data } \text{IntToInt} = \text{Mult Int} \mid \text{Add Int}\\&\text{apply } (\text{Mult } n, y) = y * n\\&\text{apply } (\text{Add } n,\ y) = y + n\\&\text{both } (f, (a, b)) = (\text{apply } (f, a), \text{apply } (f, b))\\&\text{example } (n, x) = \text{both } (\text{Mult } n, \text{both } (\text{Add } n, x))\end{aligned}$$

*Refunctionalization* is the left-inverse of defunctionalization (Danvy and Millikin 2009). It works on programs that are in the image of defunctionalization, that is, there must only be one function that pattern-matches on the algebraic data type. In that case, we can replace calls to apply by function application and constructor applications by abstractions based on the apply function and then remove the algebraic data type and the apply function. Hence we are back at the original program.

Unfortunately, refunctionalization no longer works when more than one function pattern-matches on the algebraic data type. For instance, in the defunctionalized version of the program, we can find out whether a function from Int to Int is the addition function:

$$\begin{aligned}&\text{isAdd } (\text{Add } \_) = \text{True}\\&\text{isAdd } (\text{Mult } \_) = \text{False}\end{aligned}$$

This program can no longer be refunctionalized, because there is no way to analyze a function beyond applying it to a value.

The goal of this paper is to remedy this asymmetry between defunctionalization and refunctionalization. Our main insight is that symmetry can be restored by generalizing first-class functions to codata, that is, objects defined by multiple observations (whereas functions are objects defined by just one observation, namely function application). The contributions of this paper are as follows:

- We present Uroboro, a language with pattern and copattern matching (following Abel et al. 2013), and the defunctionalization and refunctionalization between its data and codata fragments (Section 2).
- We formalize the data and codata fragments and show that the total and inverse defunctionalization and refunctionalization preserve typing and behavior (Section 3).
- We observe that the two transformations can be considered a form of matrix transposition (Section 4).
- We relate to the expression problem (Wadler 1998; Reynolds 1975; Cook 1990) by showing that the transformations switch the dimension of extensibility of the program.

Section 5 contains an extension of Reynolds's (1972) original example to demonstrate the utility of defunctionalization and unrestricted refunctionalization. We discuss our results and their relation to previous work in Section 6 and conclude in Section 7.



*ICFP'15*, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-1503-3669-7/15/08...
http://dx.doi.org/10.1145/2784731.2784763



- More examples.
- Formalization of language fragments and transformations.
- Source code of the case study.

# Relation to OO

- OO
  - codata + implicit self recursion + mutable state + …
- FP (traditional)
  - data + first-class functions
- FP (symmetric)
  - first-order functions + data + codata

# Conclusions

We generalize from first-class functions
to codata types with multiple observations
with arguments ...

- for automatic defunctionalization and refunctionalization
- to formulate (de|re)functionalization as matrix transposition that switches the dimension of extensibility
- to study one aspect of OO in an FP setting

# Conclusions

We generalize from first-class functions
to codata types with multiple observations
with arguments ...

- for automatic defunctionalization and refunctionalization
- to formulate (de|re)functionalization as matrix transposition that switches the dimension of extensibility
- to study one aspect of OO in an FP setting

*Thank you!*